Name: Luke Zhuang
Email: Lzhuang@calpoly.edu

Introduction:

This report examines the effectiveness of two approaches, Dynamic Programming and Branch and Bound, for solving the Knapsack Problem. It analyzes their performance using different test cases and explores their respective strengths, weaknesses, and potential optimizations.

Summary Table:

	Easy20	Easy50	Hard50	Easy200	Hard200
Dynamic Program	Execution time: 0.001 seconds Weight: 519 Value: 726	Execution time: 0.002 seconds Weight: 250 Value: 1123	Execution time: 0.012 seconds Weight: 10110 Value: 10110	Execution time: 0.008 seconds Weight: 2658 Value: 4092	Execution time: 0.096 seconds Weight: 112648 Value: 112648
Branch and Bound	Execution time: 0.003 seconds Weight: 519 Value: 726	Execution time: 0.004 seconds Weight: 250 Value: 1123	Execution time: 0.004 seconds Weight: 10110 Value: 10110	Execution time: 0.005 seconds Weight: 2656 Value: 4092	java.lang.OutOfMemoryError: Java heap space

Branch and Bound Hard200 Explanation:

The "java.lang.OutOfMemoryError: Java heap space" error occurs when my program **exceeds the memory** allocated to the Java Virtual Machine's heap. This is usually caused by large data structures or recursive operations. **Possible solutions** include increasing the heap size or optimizing the algorithm for better memory usage.

Dynamic Programming Approach:

A. Description of important data structures and key points of the algorithm

The key **data structure** used in our Dynamic Programming approach is a 2-dimensional integer array (or matrix). Each cell in the matrix represents the maximum value that can be obtained with a certain weight limit and a certain number of items. The rows of the matrix represent the different items, and the columns represent the different weight capacities. The **key point** is that the **recurrence relation** is:

Knapsack(i, j) = max{knapsack[i - 1, W], Knapsack[i - 1, W - Wi] + value(i)}

B. Time complexity of your implementation – theoretical and empirical

Theoretically, the time complexity of the Dynamic Programming algorithm is O(n * W), where 'n' is the number of items and 'W' is the maximum weight. This is because the algorithm has to iterate over each item for each weight capacity, thus filling up the 2-dimensional matrix.
Empirically, as the size of the problem escalates, the algorithm's time complexity appears to grow at a quadratic rate.

C. Pros and Cons

Pros:

- The Dynamic Programming algorithm is **guaranteed** to find the optimal solution
- It is **efficient** for moderately-sized instances with polynomial time complexity.

Cons:

- The algorithm's **high space complexity** (O(n * W)) is challenging for larger inputs.

D. Possible improvements

One possible improvement to the Dynamic Programming algorithm is the use of **space optimization techniques**. The algorithm can be modified to use a 1-dimensional array instead of a 2-dimensional one, reducing the space complexity to O(W).

Branch and Bound Approach:

A. Description of the search strategy and bounding function:

The **search strategy** for the Branch and Bound algorithm in this problem is implemented through a priority queue data structure (max heap), which stores the nodes of the decision tree. The node with the highest bound (estimated solution) is given the highest priority. The **bounding function** calculates an upper bound on the maximum profit from a node. If the bound is lower than the current maximum profit, the node is pruned, saving computation time.

B. Description of the data structures used and key points of the algorithm:

The Branch and Bound algorithm uses the 'Node' class to represent decision tree nodes, storing information like level, profit, weight, and upper bound. A priority queue (max heap) holds live nodes, prioritizing those with the highest bound. **Key points** involve generating child nodes (for inclusion and exclusion of the current item), calculating bounds, and deciding whether to explore or prune based on bounds and the current maximum profit.

C. Analyze and discuss both the theoretical and empirical time complexity:

Theoretical time complexity of Branch and Bound: O(n^2 * 2^n). **Empirically**, Branch and Bound performed well for smaller problems but faced memory issues for larger problems.

D. Pros and Cons

Pros:

- The Branch and Bound algorithm **can find** the optimal solution.
- It's more efficient than brute force as it **prunes non-promising nodes**.

Cons:

- It has **high memory usage** because it maintains all live nodes in memory.
- The worst-case time complexity can be **quite high**.

E. Possible improvements:

One possible improvement could be applying a more efficient bounding function that could **prune more nodes**. Also, using a **depth-first search** (DFS) strategy could reduce memory usage since it doesn't need to keep all live nodes in memory.

Conclusion:

In conclusion, the implementation of Dynamic Programming and Branch and Bound algorithms has provided valuable insights into their operational dynamics. Dynamic Programming showed efficiency in handling problems of various sizes, while facing limitations with non-integer weight capacities. Branch and Bound excelled in smaller problems but encountered memory issues with larger ones. Despite their limitations, both algorithms hold significant potential in solving complex computational problems.